Extracted from:

Hello, Android

Introducing Google's Mobile Development Platform, 3rd Edition

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Updated for Kindle Fire

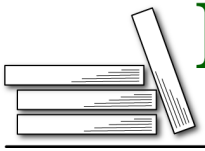
Hello, Android

Introducing Google's Mobile Development Platform

Third Edition

Ed Burnette

Edited by Susannah Davidson Pfalzer



Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

Gesture icons in Chapter 11 courtesy of GestureWorks (www.gestureworks.com).

The team that produced this book includes:

Susannah Davidson Pfalzer (editor)
Seth Maislin (indexer)
Kim Wimpsett (copyeditor)
David Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-934356-56-2
Printed on acid-free paper.
Book version: P8.0—January 2012

Over the next few chapters, we'll cover more advanced topics such as network access and location-based services. You can write many useful applications without these features, but going beyond the basic features of Android will really help you add value to your programs, giving them much more functionality with a minimum of effort.

What do you use your mobile phone for? Aside from making calls, more and more people are using their phones as mobile Internet devices. Analysts predict that in a few years mobile phones will surpass desktop computers as the number-one way to connect to the Internet.[1] This point has already been reached in some parts of the world.[2]

Android phones are well equipped for the new connected world of the mobile Internet. First, Android provides a full-featured web browser based on the WebKit open source project.[3] This is the same engine you will find in Google Chrome, the Apple iPhone, and the Safari desktop browser but with a twist. Android lets you use the browser as a component right inside your application.

Second, Android gives your programs access to standard network services like TCP/IP sockets. This lets you consume web services from Google, Yahoo, Amazon, and many other sources on the Internet.

In this chapter, you'll learn how to take advantage of all these features and more through four example programs:

- *BrowserIntent*: Demonstrates opening an external web browser using an Android intent
- *BrowserView*: Shows you how to embed a browser directly into your application
- *LocalBrowser*: Explains how JavaScript in an embedded `WebView` and Java code in your Android program can talk to each other
- *Suggest*: Uses data binding, threading, and web services for an amusing purpose

7.1 Browsing by Intent

The simplest thing you can do with Android's networking API is to open a browser on a web page of your choice. You might want to do this to provide a link to your home page from your program or to access some server-based

1. http://archive.mobilecomputingnews.com/2010/0205.html
2. http://www.comscore.com/press/release.asp?press=1742
3. http://webkit.org

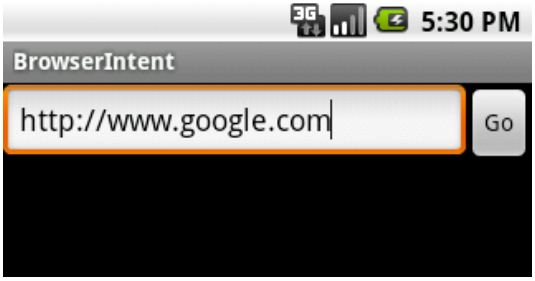


Figure 34—Opening a browser using an Android intent

application such as an ordering system. In Android all it takes is three lines of code.

To demonstrate, let's write a new example called BrowserIntent, which will have an edit field where you can enter a URL and a Go button you press to open the browser on that URL (see Figure 34, *Opening a browser using an Android intent*, on page 6). Start by creating a new "Hello, Android" project with the following values in the New Project wizard:

```
Project Name: BrowserIntent
Build Target: Android 2.3.3
Application Name: BrowserIntent
Package Name: org.example.browserintent
Create Activity: BrowserIntent
Minimum SDK: 10
```

Once you have the basic program, change the layout file (res/layout/main.xml) so it looks like this:

Download BrowserIntent/res/layout/main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="horizontal"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent">
   <EditText
      android:id="@+id/url_field"
      android:layout_width="0dip"
      android:layout_height="wrap_content"
      android:layout_weight="1.0"
      android:lines="1"
      android:inputType="textUri"
      android:imeOptions="actionGo" />

   <Button
      android:id="@+id/go_button"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/go_button" />
</LinearLayout>
```

This defines our two controls, an EditText control and a Button.

On EditText, we set android:layout_weight="1.0" to make the text area fill up all the horizontal space to the left of the button, and we also set android:lines="1" to limit the height of the control to one vertical line. Note that this has no effect on the amount of text the user can enter here, just the way it is displayed.

Android 1.5 introduced support for soft keyboards and other alternate input methods. The options for android:inputType="textUri" andandroid:imeOptions="actionGo" are hints for how the soft keyboard should appear. They tell Android to replace the standard keyboard with one that has convenient buttons for ".com" and "/" to enter web addresses and has a Go button that opens the web page.[4]

As always, human-readable text should be put in a resource file, res/values/strings.xml.

Download BrowserIntent/res/values/strings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <string name="app_name">BrowserIntent</string>
   <string name="go_button">Go</string>
</resources>
```

4. See http://d.android.com/reference/android/widget/TextView.html and http://android-developers.blogspot.com/2009/04/updating-applications-for-on-screen.html for more information on input options.

Next we need to fill in the onCreate() method in the BrowserIntent class. This is where we'll build the user interface and hook up all the behavior. If you don't feel like typing all this in, the complete source code is available online at the book's website.[5]

Download BrowserIntent/src/org/example/browserintent/BrowserIntent.java

```
Line 1 package org.example.browserintent;
  - import android.app.Activity;
  - import android.content.Intent;
  - import android.net.Uri;
  5 import android.os.Bundle;
  - import android.view.KeyEvent;
  - import android.view.View;
  - import android.view.View.OnClickListener;
  - import android.view.inputmethod.EditorInfo;
 10 import android.view.inputmethod.InputMethodManager;
  - import android.widget.Button;
  - import android.widget.EditText;
  - import android.widget.TextView;
  - import android.widget.TextView.OnEditorActionListener;
 15 public class BrowserIntent extends Activity {
  -    private EditText urlText;
  -    private Button goButton;
  -    @Override
  -    public void onCreate(Bundle savedInstanceState) {
 20       super.onCreate(savedInstanceState);
  -       setContentView(R.layout.main);
  -       // Get a handle to all user interface elements
  -       urlText = (EditText) findViewById(R.id.url_field);
  -       goButton = (Button) findViewById(R.id.go_button);
 25       // Setup event handlers
  -       goButton.setOnClickListener(new OnClickListener() {
  -          public void onClick(View view) {
  -             openBrowser();
  -          }
 30       });
  -       urlText.setOnEditorActionListener(new OnEditorActionListener() {
  -          public boolean onEditorAction(TextView v, int actionId,
  -                KeyEvent event) {
  -             if (actionId == EditorInfo.IME_NULL) {
 35                openBrowser();
  -                InputMethodManager imm = (InputMethodManager)
  -                      getSystemService(INPUT_METHOD_SERVICE);
  -                imm.hideSoftInputFromWindow(v.getWindowToken(), 0);
  -                return true;
 40             }
  -             return false;
  -          }
```

5. http://pragprog.com/titles/eband3

```
        });
    }
}
```

Inside onCreate(), we call setContentView() on line 21 to load the view from its definition in the layout resource, and then we call findViewById() on line 23 to get a handle to our two user interface controls.

Line 26 tells Android to run some code when the user selects the Go button, either by touching it or by navigating to it and pressing the center D-pad button. When that happens, we call the openBrowser() method, which will be defined in a moment.

If the user types an address and hits the Enter key (or the Go button on the soft keyboard), we also want the browser to open. To do this, we define a listener starting on line 31 that will be called when an action is performed on the edit field. If the Enter key or Go button is pressed, then we call the openBrowser() method to open the browser; otherwise, we return false to let the edit control handle the event normally.

Before returning, we also get a handle to the system's InputMethodManager and call the hideSoftInputFromWindow() method in order to close the soft input window, because it has an annoying habit of staying open when we don't need it any more.

Now comes the part you've been waiting for: the openBrowser() method. As promised, it's three lines long:

Download BrowserIntent/src/org/example/browserintent/BrowserIntent.java

```
/** Open a browser on the URL specified in the text box */
private void openBrowser() {
   Uri uri = Uri.parse(urlText.getText().toString());
   Intent intent = new Intent(Intent.ACTION_VIEW, uri);
   startActivity(intent);
}
```

The first line retrieves the address of the web page as a string (for example, "http://www.android.com") and converts it to a uniform resource identifier (URI).

Note: Don't leave off the "http://" part of the URL when you try this. If you do, the program will crash because Android won't know how to handle the address. In a real program you could add that if the user omitted it.

The next line creates a new Intent class with an action of ACTION_VIEW, passing it the Uri class just created as the object we want to view. Finally, we call the startActivity() method to request that this action be performed.

When the Browser activity starts, it will create its own view (see Figure 35, *Viewing a web page with the default browser*, on page 11), and your program will be paused. If the user presses the Back key at that point, the browser window will go away, and your application will continue. But what if you want to see some of your user interface and a web page at the same time? Android allows you to do that by using the WebView class.

7.2 Web with a View

On your desktop computer, a web browser is a large, complicated, memory-gobbling program with all sorts of features like bookmarks, plug-ins, Flash animations, tabs, scroll bars, printing, and so forth.

When I was working on the Eclipse project and someone suggested replacing some common text views with embedded web browsers, I thought they were crazy. Wouldn't it make more sense, I argued, to simply enhance the text viewer to do italics or tables or whatever it was that was missing?

It turns out they weren't crazy because:

- A web browser can be (relatively) lean and mean if you strip out everything but the basic rendering engine.
- If you enhance a text view to add more and more things that a browser engine can do, you end up with either an overly complicated, bloated text viewer or an underpowered browser.

Android provides a wrapper around the WebKit browser engine called WebView that you can use to get the real power of a browser with as little as 1MB of overhead. Although 1MB is still significant on an embedded device, there are many cases where using a WebView is appropriate.

WebView works pretty much like any other Android view except that it has extra methods specific to the browser. I'm going to show how it works by doing an embedded version of the previous example. This one will be called BrowserView instead of BrowserIntent, since it uses an embedded View instead of an Intent. Start by creating a new "Hello, Android" project using these settings:

```
Project Name: BrowserView
Build Target: Android 2.3.3
Application Name: BrowserView
Package Name: org.example.browserview
Create Activity: BrowserView
Minimum SDK: 10
```

The layout file for BrowserView is similar to the one in BrowserIntent, except we've added a WebView at the bottom:

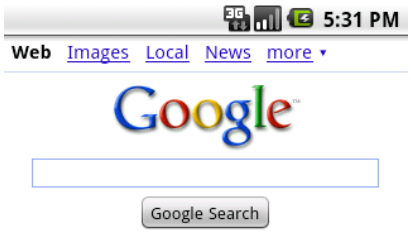


Figure 35—Viewing a web page with the default browser

Download BrowserView/res/layout/main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent">
   <LinearLayout
      android:orientation="horizontal"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content">
      <EditText
         android:id="@+id/url_field"
         android:layout_width="0dip"
         android:layout_height="wrap_content"
         android:layout_weight="1.0"
         android:lines="1"
         android:inputType="textUri"
         android:imeOptions="actionGo" />
      <Button
         android:id="@+id/go_button"
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:text="@string/go_button" />
   </LinearLayout>
   <WebView
      android:id="@+id/web_view"
      android:layout_width="fill_parent"
      android:layout_height="0dip"
      android:layout_weight="1.0" />
</LinearLayout>
```

We use two LinearLayout controls to make everything appear in the right place. The outermost control divides the screen into top and bottom regions; the top has the text area and button, and the bottom has the WebView. The innermost LinearLayout is the same as before; it just makes the text area go on the left and the button on the right.

Joe asks:
Why Didn't BrowserIntent Need <uses-permission>?

The previous example, BrowserIntent, simply fired off an intent to request that some other application view the web page. That other application (the browser) is the one that needs to ask for Internet permissions in its own AndroidManifest.xml.

The onCreate() method for BrowserView is exactly the same as before, except that now there is one extra view to look up:

Download BrowserView/src/org/example/browserview/BrowserView.java

```
import android.webkit.WebView;
// ...

public class BrowserView extends Activity {
   private WebView webView;
   // ...
   @Override
   public void onCreate(Bundle savedInstanceState) {
      // ...
      webView = (WebView) findViewById(R.id.web_view);
      // ...
   }
}
```

The openBrowser() method, however, is different:

Download BrowserView/src/org/example/browserview/BrowserView.java

```
/** Open a browser on the URL specified in the text box */
private void openBrowser() {
   webView.getSettings().setJavaScriptEnabled(true);
   webView.loadUrl(urlText.getText().toString());

}
```

The loadUrl() method causes the browser engine to begin loading and displaying a web page at the given address. It returns immediately even though the actual loading may take some time (if it finishes at all).

Don't forget to update the string resources:

Download BrowserView/res/values/strings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <string name="app_name">BrowserView</string>
   <string name="go_button">Go</string>
</resources>
```

We need to make one more change to the program. Add this line to AndroidManifest.xml before the <application> tag:

Download BrowserView/AndroidManifest.xml

```
<uses-permission android:name="android.permission.INTERNET" />
```

If you leave this out, Android will not give your application access to the Internet, and you'll get a "Web page not available" error.

Try running the program now, and enter a valid web address starting with "http://"; when you press Return or select the Go button, the web page should appear (see Figure 36, *Embedding a browser using WebView*, on page 14).

WebView has dozens of other methods you can use to control what is being displayed or get notifications on state changes.

You can find a complete list in the online documentation for WebView, but here are the methods you are most likely to need:

- addJavascriptInterface(): Allows a Java object to be accessed from JavaScript (more on this one in the next section)
- createSnapshot(): Creates a screenshot of the current page
- getSettings(): Returns a WebSettings object used to control the settings
- loadData(): Loads the given string data into the browser
- loadDataWithBaseURL(): Loads the given data using a base URL
- loadUrl(): Loads a web page from the given URL
- setDownloadListener(): Registers callbacks for download events, such as when the user downloads a .zip or .apk file
- setWebChromeClient(): Registers callbacks for events that need to be done outside the WebView rectangle, such as updating the title or progress bar or opening a JavaScript dialog box
- setWebViewClient(): Lets the application set hooks in the browser to intercept events such as resource loads, key presses, and authorization requests
- stopLoading(): Stops the current page from loading

One of the most powerful things you can do with the WebView control is to talk back and forth between it and the Android application that contains it. Let's take a closer look at this feature now.

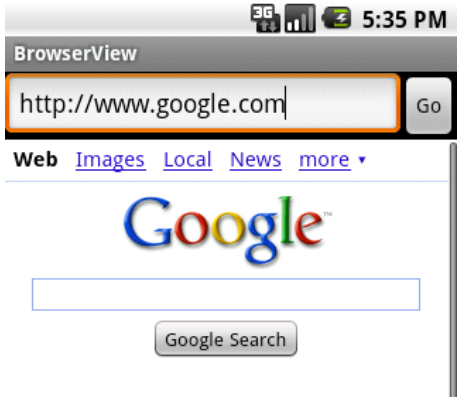


Figure 36—Embedding a browser using WebView